Static Scheduling of Object-Based Real-time Tasks with Probabilistic Conditional Branches in Distributed Systems

I. Santhoshkumar G. Manimaran C. Siva Ram Murthy
Department of Computer Science & Engineering
Indian Institute of Technology
Madras, 600 036, INDIA
{{skumar,gmani}@bronto.,murthy@}iitm.ernet.in

Abstract

In this paper, we propose an algorithm for scheduling object-based real-time tasks having probabilistic conditional branches, where the probability of executing every branch of the conditionals is known *a priori*, onto the processing elements (with necessary communication channel scheduling) of a distributed system. In addition, the tasks can have precedence and resource constraints among them. Since the schedule should allow any execution instance (path) to be executed at run-time and the number of execution instances may be exponential, scheduling of tasks corresponding to this task model is a difficult problem. Our algorithm attempts to find a final unified schedule by merging the schedules obtained for a selected set of execution instances which cover all possible executable codes in all the tasks. The selection of execution instances is based either on probabilities or on deadlines associated with them, and scheduling of these execution instances is done using a deterministic scheduling algorithm which exploits the parallelism inherent among the components of object-based tasks using ARPCs and cloning. The merging is based on a weight associated with each of these individual schedules. The weight can either be the probability or the criticality (deadline) of the execution instance, or a combination of both. We have conducted extensive simulations to study the effectiveness of our algorithm by comparing it with the existing algorithms. Our simulation study reveals that the selection of execution instances based on deadline together with merging based on both deadline and probability offers the best schedulability.

1 Introduction

In real-time applications, the correctness of computation depends not only on the logical results of computation but also the time at which the results are generated. Air traffic control and process control are some examples of present real-time systems. Such systems are life-critical and the outcome would be catastrophic, if the results are not generated within specified time intervals. Current real-time computing technology is challenged to handle more sophisticated and complex systems such as space stations, automated factories, and advanced command and control systems. Most of these emerging real-time systems are large, highly parallel, and distributed. The complexity in the development of software for such systems can be managed by using object-based design and methodology [1]. Even though the reusable software components contained in the object-based implementation of an application have advantages such as abstraction and encapsulation [2], execution efficiency may have to be sacrificed due to the large number

of procedure calls and contention for accessing shared software components [3]. Another problem that plagues the scheduling of object-based tasks, on a distributed system, is the presence of conditional branches in the tasks which may have probabilities associated with each branch denoting the likelihood that the branch will be executed at run-time. The major difficulty in scheduling conditional branches is that the direction of each branch will be known only at run-time [4]. Also, the presence of an exponential number of paths in each task due to conditional branches makes the scheduling problem more complex. In this paper, we propose an algorithm for scheduling object-based tasks with probabilistic conditional branches in a distributed real-time system.

The rest of the paper is organized as follows: In Section 2, the probabilistic periodic task model considered by our scheduling algorithm is described. The existing literature in pre-run-time scheduling algorithms which schedule object-based tasks is discussed in Section 3. Section 4 describes our algorithm for scheduling object-based tasks having conditional branches in it. The experimental evaluation of the algorithm is presented in Section 5. Finally, in Section 6 we make some concluding remarks.

2 System and Task Model

We consider a distributed system having n processing elements. The processing elements are connected by a multiple-access network employing a TDMA protocol. Thus, it is possible to predict when a message will be delivered/accepted. The system is assumed to have a set of resources. Each resource is attached to a particular processing element and any access to that resource has to be made through the corresponding processing element.

We consider *probabilistic model*, where object-based tasks will contain conditional branches with probabilities assigned to the branches. The application is designed in such a way that there are t periodic tasks $T = \{T_1, T_2, \ldots, T_t\}$ where the structure of a task is as shown in figure 1. Each task T_i is assumed to be periodic with period P_i (and the computation time of the task is defined at the bead level) and its deadline is equal to its period.

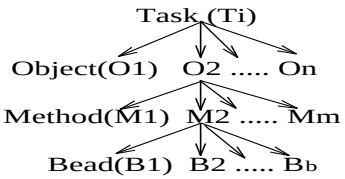

Figure 1 Task Model

Each task will contain a set of objects which are the instances of software components spread over a set of programs. The software components are either *abstract data types* (ADTs) or *abstract data objects* (ADOs) [1]. Each software component or object will have a set of methods operating on the data encapsulated in the object. At run-time, the objects communicate with each other through method calls. If the caller and callee are assigned to the same processor, then the method call can be implemented by a *local procedure call* (LPC). In the other case, when the caller and callee are on different processing elements it is implemented by a remote procedure call (RPC). RPC can be either *asynchronous* (ARPC) or *synchronous* (SRPC). SRPCs are those method calls

in which the caller gets blocked after the call. With ARPC, the caller can continue its execution in parallel with the callee till the caller needs the results from the callee. By doing static analysis on the object-based code, the procedure calls which can be implemented by ARPC can be identified.

The points at which external procedure calls are made are known as *preemption points*. It is according to the semi-preemption model proposed in [1], where tasks are preempted only at predefined preemption points such as external procedure calls and blocking device accesses. Two consecutive preemption points constitute a non-preemptable entity called a *bead*. Beads can be of two types, computation beads and communication beads. A computation bead is the execution of sequential code while the communication between two methods constitute a communication bead. Thus, every method is the combination of one or more computation beads.

For each computation (communication) bead B, $E(B)$ denotes the computation (communication) time of the bead. Since the beads constitute a method, $E(M)$ stands for the sum of the computation times of all the beads of the method M. $NM(M)$ denotes the total number of calls made to M by all tasks for all periods up to the least common multiple (LCM) of task periods (also known as *planning cycle*). The execution time of a task T_i is the sum of the execution times of all the methods present in the implementation of the task which is denoted by $E(T_i)$.

Conditionals make the task execution to go through one of the several possible paths. The probability of executing each path is known prior to run-time. It can be obtained by running the program several times for different set of inputs. The *split* and *join* of the task at conditionals are represented by double lined edges called *branching edges*. Split edges are those where a conditional branching occurs. The different branches of a conditional join together through join branching edges. The probability $P(B_i, B_j)$ associated with the split branching edge from bead B_i to bead B_j denotes the probability of executing B_j after B_i. For any B_i, summation of the probabilities $P(B_i, B_j)$ associated with each split branching edge (B_i, B_j) is equal to one, i.e., $\sum P(B_i, B_j) = 1$, where (B_i, B_j) is a split branching edge. The probability associated with each join branching edge is equal to 1, since the start of a conditional implies a definite end to it. Associated with each bead B_j there is a probability denoted by $OP(B_j)$ which stands for the probability that B_j will be executed. $OP(B_j)$ for a bead B_j is defined as $OP(B_j) = \sum P(B_i, B_j) * OP(B_i)$ where B_i is an immediate predecessor of B_j.

The sequence of beads of each task that will be executed for an input constitute an *execution instance* (EI). Each EI will contain a certain path (or flow of control) from each of the object-based tasks. Each path of a task T_i is assumed to have the same period and its deadline is equal to the task period P_i. The probability of a path to be included in an EI is equal to the product of the probabilities associated with every branching edge present in the path. The probability of an execution instance E_i to be generated, denoted as $Pr(E_i)$, is the product of the probabilities associated with every path present in E_i.

Resource requirements of tasks are represented at method level. If a task needs to access a resource, a method of the environment dependent software component has to be called. If two different tasks try to call a method of an environment dependent software component, then one of them has to wait. Thus, resource constraints are modelled in the same way as the access to a shared component. Thus, the execution of an *object-based task* will be spanning over a set of programs where it starts from one program,

switches to other programs through method calls, and branches at conditionals with certain probabilities.

3 Earlier Work

A model for pre-run-time scheduling of object-based distributed real-time systems that are composed of ADTs and ADOs is proposed in [1, 5]. In addition, they present an incremental scheduling approach which constructs an initial schedule and modifies it by enhancing concurrency through ARPC and cloning, till a feasible schedule is obtained. The work in [3] presents compiler techniques for identifying concurrency among software components via ARPCs and cloning in the context of an incremental scheduling algorithm. Difference between ours and their work is that our model considers periodic tasks having precedence constraints among them compared to the multiple independent tasks considered in [3]. In [9], we have proposed an algorithm to schedule object-based tasks, having precedence and resource constraints among them, onto the processing elements of a distributed system with necessary communication channel scheduling.

The problem of allocating task graphs that contain conditional branches for non-real-time systems was discussed in [7, 8]. An algorithm based on the shortest path method to allocate distributed programs was presented in [7]. The work in [8] considered a computational model that includes conditional branches and concurrent execution of program modules. Recently, El-Rewini and Ali [4] have proposed a static algorithm for scheduling non-real-time tasks having probabilistic conditional branches in it. The aim of their scheduling algorithm was to reduce the schedule length which is not a metric in real-time systems. The work in [4] constructs a unified schedule by merging the schedules of a small number of most probable execution instances. The merging in [4] was based on *weights*, which are the probabilities of different execution instances to be generated.

The model presented in [1] considered conditional branches in real-time tasks by making those beads that belong to different branches of a conditional to be independent. Their algorithm tries to overlap such beads in time if they are assigned to the same processing element. Most importantly, it does not exploit the probability information in order to improve the schedulability. But, our model considers the probability of execution of each conditional branch and exploits this probability information to improve the schedulability. The recent work in [6], considers the probabilities assigned to the branches of conditionals to calculate processor utilization which guides the assignment and scheduling of tasks. But, our approach is guided by the probability and criticality associated with each branch of conditionals, where parallelism involved in each branch is exploited using ARPCs and cloning.

4 Our Scheduling Algorithm

Our scheduling approach has three steps which are similar to the approach used in [4]. The basic idea of the three steps is to get a final unified schedule by merging the schedules generated from a selected set of execution instances (EIs). The essence of each of the steps is explained below. The detailed explanation of the steps will follow that.

1. Generating EIs: If the total number of possible EIs is not too large, all the EIs are generated. The total number of EIs is equal to the product of the number of paths in each

of the tasks. If it is exponential, then filtering has to be done to generate candidate EIs. The basic criteria of this step is that each bead of every task must be included at least in one EI. There are two ways of generating execution instances: (a) probability based [4] and (b) critical path based. Probability based EI generation generates the highest probable execution instances, whereas the critical path instance generation produces EIs having the highest computation times.

2. Producing Schedule for Each Execution Instance: This step is to construct a schedule for each generated EI using a deterministic pre-run-time scheduling algorithm. Each schedule will have a *weight* associated with it depending on the type of EI generation.

3. Merging the Produced Schedules: The final schedule is constructed by merging the schedules produced in step 2. The methods and beads are allocated and scheduled on different processing elements based on the weights of the schedules merged. The weights that we consider are: (a) the probability of realizing execution instances at run-time, (b) criticality of execution instances, and (c) a combination of both the probability and criticality of execution instances.

4.1 Generating EIs

As discussed earlier, execution instances can be generated either based on probabilities or critical paths. It is assumed that number of possible EIs are exponential. Otherwise every EI will be generated. Let the EIs generated be $SE = \{E_1, E_2, \ldots, E_n\}$, where each E_i consists of a set of beads $BEADS(E_i) = \{B_{i1}, B_{i2}, \ldots, B_{im}\}$ such that $\bigcup_{i=1}^{n} BEADS(E_i)$ covers all the beads among all the tasks. For each bead B_i, $IMS(B_i)$ denotes the immediate successors of B_i. Here, we present the critical path based EI generation. The probability based generation of EIs is proposed in [4].

Critical Path Based EI Generation: The *critical path* in a task is defined as the path in the task having the largest computation time, which is the same as the path having the highest criticality (i.e., least laxity). The basic criterion for EI generataion is that every bead in each task must be included at least in one of the EIs generated.

```
Algorithm GenerateEICritical
/* SE denotes the set of EIs and E denotes each distinct EI */
Initially all beads are unmarked
SE = ∅
while (unmarked beads are present)
    E = ∅
    for (every task)
        if (all beads are marked)
            P = Path which has the largest computation time among the marked beads
            E = E ⋃ beads in path P
        else
            P = Path which has the largest computation time among the unmarked beads
            E = E ⋃ beads in path P
            mark the beads in P
        endif
    end for
    SE = SE ⋃ E
end while
```

All the tasks need not have the same number of execution paths. So the task with less number of paths will have all its beads marked before another task having more number of paths. Since all the beads in every task have to be included in at least one execution instance and each execution instance will have one path from each task, the

paths in the task with less number of paths will be repeated. That is why the *if* and *else* statements consider a successor from *marked* and *unmarked* beads, respectively (see the algorithm). For example, let T_1 has paths $\{P_{11}, P_{12}\}$ and T_2 has paths $\{P_{21}, P_{22}, P_{23}\}$ where each path P_{ij} contains beads that will execute for a particular input in task T_i. Let the criticality of P_{ij} be greater than $P_{i(j+1)}$. Then $E_1 = \{P_{11}, P_{21}\}$ and $E_2 = \{P_{12}, P_{22}\}$. Since all the paths (hence all the beads) in T_1 are exhausted and there is one more path (hence some more beads) in T_2 to be included, one more EI has to be generated which will contain the nonmarked path P_{23} in T_2 and the highest critical marked path in T_1, which is P_{11}. Hence, $E_3 = \{P_{11}, P_{23}\}$.

If the highest probable path is assigned to the variable P in the algorithm *GenerateEICritical* rather than the path with the highest computational requirements, the most probable execution instances can be generated as in [4].

4.2 Scheduling of Each Execution Instance

A deterministic scheduling algorithm is used to schedule each selected execution instance. The deterministic scheduling algorithms proposed in [9] can be used to schedule each execution instance. The algorithms in [9] employ parallelization exploitation techniques such as ARPCs and cloning to enhance schedulability.

The schedule generated for each execution instance has a *weight* associated with it. Thus, the weight of a schedule SC_i is denoted by $W(SC_i)$. If an execution instance E_i is generated based on probability, then the weight of the corresponding schedule is equal to $Pr(E_i)$ [4]. If an execution instance E_i is generated based on its criticality (i.e., critical path) rather than probability, then the weight of the corresponding schedule is a quantified measure of its criticality. It can be calculated as follows. Let the sum of the computation time of the beads in the (most) critical path of a task T_i is denoted by $MCP(T_i)$ and the sum of the computation time of the beads of a task T_i in each generated execution instance E_j is denoted by $CP(T_i, E_j)$. Then, the weight of the schedule for an execution instance E_j is $W(SC_j) = \prod_{i=1}^{t} CP(T_i, E_j)/MCP(T_i)$. Another possibility is to associate weights to the schedules created for the most critical execution instances based on their criticality and probability.

4.3 Merging the Schedules

This step has mainly three substeps: (a) method allocation (b) bead ordering, and (c) bead scheduling. Here, *allocation, ordering,* and *scheduling* refer to the final unified schedule. In a method level allocation, all the beads of a method must execute on the same processing element. Since the execution of the beads of different methods can be interleaved, ordering has to be done at bead level. Finally, each bead is assigned a start time (scheduling) without violating its precedence, resource, and periodicity constraints.

Method Allocation: This step decides the processing element to which a method will be allocated in the final schedule. The allocation of method to a processing element is decided based on the summation of the weights of the schedules to be merged and the number of times a method is allocated to that processing element.

Algorithm MethodAllocation
/* There are S schedules $SC_1, SC_2, \ldots, SC_S$ to be merged */
Let $N_p(M_i)$ denote the number of times M_i is allocated to processor p.

Let $Maj_p(M_i) = \sum_{x=1}^{S} W(SC_x)$ if M_i is allocated to processing element p in schedule SC_x.
Method M_i is allocated to that processing element for which (Maj_p, N_p) is the highest in lexicographic order.

Bead Ordering: The execution order of two beads is determined by the order in which they appear in those schedules in which they are assigned to the same processing element. If there exist two different orders for a pair of beads, then the summation of the weights of the schedules associated with each order is used to decide the final ordering. For example, let the order between B_i and B_j is to be determined. There are 4 schedules to be merged and B_i is scheduled before B_j in two schedules having weights a and b, respectively. In the other two schedules B_j comes before B_i and the weight of those schedules are c and d. Let $(a+b)$ and $(c+d)$ be greater than zero. In the final schedule B_i will be scheduled before B_j, provided $(a+b) > (c+d)$. If $(c+d) > (a+b)$ then B_j will be scheduled before B_i. If those two beads do not exist on the same processing element in any of the schedules, then their order of execution depends on the presence of any precedence relations between them. If it is not possible to decide their order by the above checks, it is determined randomly.

```
Algorithm BeadOrdering
/* There are S schedules SC_1, SC_2, ..., SC_S to be merged */
for (each pair of beads (B_i, B_j) such that they are allocated to the same processing element)
    MajWt = 0
    for (every schedule SC_g)
        if (B_i and B_j are allocated to the same processing element p in SC_g)
            if (B_i comes before B_j in SC_g)
                MajWt = MajWt + W(SC_g)
            else
                MajWt = MajWt - W(SC_g)
            end if
            if ((MajWt > 0) or (B_i precedes B_j))
                B_i precedes B_j in final schedule
            end if
            if ((MajWt < 0) or (B_j precedes B_i))
                B_j precedes B_i in final schedule
            end if
            if (B_i and B_j are not yet ordered)
                Pick the order randomly
            end if
        end if
    end for
end for
```

Bead Overlapping and Scheduling The next step is to find the start time of each bead which is decided in such a way that the precedence and periodicity constraints are not violated. The latest start time and earliest start time (EST) of each bead is found from the original task graph which contains conditional branches. EST of a bead is the earliest time at which it can start. The start time assigned to each bead must be greater than or equal to its EST and less than or equal to its LST. If two or more beads (which correspond to different branches of a conditional) are consecutively ordered on a processing element, then they can be overlapped.

5 Simulation Study

To study the effectiveness of our scheduling algorithm, we have conducted extensive simulation studies. The performance metric is the *success ratio* which is defined as

the number of task sets found schedulable by an algorithm to the number of task sets considered for scheduling. Each point in the performance curves (figures 2-5) is the average of 5 simulation runs each with 100 schedulable periodic task sets. The parameters used in our simulation are given in the following table. The task sets are generated based on the following:

• Suppose there are 5 periodic tasks having $x, 2x, 3x, 4x$, and $6x$ methods, respectively, which adds to $16x$ methods. The actual number of methods in all software components is $M = MToMRatio * 16x$, where $MToMRatio$ (method to method ratio) is the ratio of the total number of methods in all software components to the total number of methods called in one EI over all tasks. It means that a method may be called more than once. Total number of software components is $Sof\_Ratio * M$, where Sof_Ratio determines the sharing of software components.

• All computation and communication costs are in the same units. The computation time of a bead is chosen between 30 and 50 units and communication time is chosen less than or equal to $CommRatio * C$, where C is the average computation time (40). All the results shown here are for $CommRatio = 0.4$.

• The period of each task is chosen based on the critical path in the task. The period of the first task (with x methods) is chosen as $P = (1 + C * LaxityFactor * Number\_of\_Beads\_in\_CriticalPath)$ where $Number\_of\_Beads\_in\_CriticalPath$ is the total number of computation beads in the critical path of the task. The periods of other tasks are equal to 2P, 3P, 4P, and 6P. Thus, the planning cycle remains 12P. The tasks generated must have probabilistic conditional branches in it.

• The parameter Non_Det decides the number of beads that will be present in the branches of conditionals. It is the ratio of the number of beads with probabilities less than one to the total number of beads. The probabilities associated with each branch are determined randomly.

parameter	explanation
Num_Proc	the number of processors in the system
LaxityFactor	denotes the tightness of the deadline
Num_Sof	number of software components to allocate
Arpc_Prob	probability for a method call to be an ARPC
Env_Prob	probability for a software component to represent a resource
Sof_Ratio	determines the sharing of software components
Non_Det	determines the number of beads in conditional branches

Based on the scheme of generation of EIs and the weight used in the merging step, we have three *conditional branching algorithms*: (1) Probability based EI generation and probability based merging (in the future discussions we call it as *probability algorithm*), (2) Critical path based EI generation and criticality based merging (we call it as *critical path algorithm*), and (3) Critical path based EI generation and merging based on the combination of probability and criticality (we call it as *critical path/probability algorithm*).

The probability algorithm is similar to the work in [4]. The second algorithm namely, critical path algorithm, does not take into account the probabilities associated with conditional branches. This can be viewed as the approach proposed in [1], which tries to overlap the beads of a conditional scheduled on a processing element in time. Our work is the third one, which merges the schedules of the most critical execution instances based on their criticality and probability. In all the above three algorithms, the third step involves the employment of a deterministic scheduling algorithm for which we used

an algorithm proposed in [9] (which applies clustering at the method level and performs cloning). We consider one more algorithm for comparison study which is *trivial* in the sense that it just schedules the beads on the processing elements without considering the conditional branches and also does not try for bead overlapping. The *trivial* algorithm is a deterministic algorithm, which takes the task graph containing conditional branches, forgets the presence of conditional branches, and schedules every bead assuming that all of them will be executed for every input. The deterministic algorithm used is the algorithm proposed in [9] which applies the clustering heuristic at method level and performs cloning.

In all the studies (figures 2-5), it can be observed that the trivial algorithm performs very poorly, since it will not exploit the properties of conditional branches. In other words, it does not try to overlap in time the beads that belong to different branches of conditionals. All other algorithms try to overlap the beads that belong to the different branches of conditionals during the merging step. It is based on the weight which is used for merging. The probability algorithm generates more number of feasible schedules compared to the trivial algorithm, since the probability algorithm overlaps beads in time, based on the probability that such beads will be executed at run-time. Critical path algorithm gives much better results compared to probability algorithm, since it overlaps beads in time, based on their criticality (in other words, deadline) which is more important in real-time systems. As in critical path based algorithm, our algorithm also considers the criticality of the beads while they are overlapped in time. In addition, our algorithm considers the probability associated with each critical execution instance whose schedules are merged. It can be observed from figures 2-5 that considering both criticality and probability improves the schedulability. The curves corresponding to trivial algorithm are flat since it does not consider the probability (or non-determinism) associated with the conditional branches.

Effect of Generation of Execution Instances: It can be observed from figures 2-5 that the conditional branch scheduling algorithm based on generating critical execution instances performs better than the conditional branch scheduling algorithm based on generating the most probable execution instances. It happens since the critical execution instances consider the highest computational requirements whereas the probability based execution instance generation does not take the computational requirements into account. This may result in not generating execution instances which have higher computational requirements (tighter laxities) whose deadlines may not be met.

Effect of Non-determinism: In figures 2 and 3, we study the effect of non-determinism on the performance of different conditional branching algorithms. With increase in the value of the parameter Non_Det, the number of beads that belong to the branches of conditionals increases. This increases the non-determinism inherent in the tasks. Thus, more beads can be overlapped which is exploited by our merging algorithm in order to increase the schedulability. Figure 2 corresponds to the study conducted for LaxityFactor=1.1, while figure 3 corresponds to LaxityFactor=1.2. With increase in the laxity of the object-based tasks, the conditional branch scheduling algorithms find more number of feasible schedules.

Effect of Number of Execution Instances Merged: The execution instance generation algorithms discussed in section 4.1 generate the minimum number of execution instances that will cover all the beads in every task. Here, we study the effect of

merging the schedules of more number of execution instances. The additional execution instances can either be generated randomly on the basis of their probability or on the basis of their criticality. In our study, for the algorithm which generates the highly probable execution instances, the additional instances generation is based on probability. That is, the next highly probable execution instances are generated. Similarly, the next highly critical execution instances are generated in the algorithm which generates execution instances based on criticality. It was observed that the algorithm which merges more schedules generated from the critical execution instances gives better schedulability compared to the scheme which merges more number of highly probable schedules. It is shown in figures 4 and 5 for LaxityFactor equal to 1.1 and 1.2, respectively. It is due to the fact that critical execution instance generation is based on the time criticality of the beads in each task. While more number of critical execution instances are considered, the beads in those execution instances having critical timing requirements will get more weight in the merging step, and their allocation and scheduling in the final schedule will be more *correct*. By a *correct* allocation and scheduling, we mean that allocation and scheduling, which may lead to a feasible schedule. But, considering more number of probable execution instances may not give the beads enough weight to make the allocation and scheduling in the final schedule to be correct since they are based on probabilities rather than computational requirements.

6 Conclusions

In this paper, we have proposed an algorithm to schedule object-based tasks having probabilistic conditional branches. The difficulties in scheduling tasks with conditional branching are (a) exponential number of execution instances and (b) lack of *a priori* knowledge of the direction of each branch. Our algorithm tries to find a final unified schedule for all the different execution instances by merging the schedules of a few selected set of execution instances. The scheduling of each execution instance involves exploitation of the parallelism inherent among the different components of execution instances using techniques such as ARPC and cloning. We have compared the schedulability offered by our algorithm with that of the existing algorithms which either use only the criticalities or probabilities of execution instances. We have shown through simulation that the selection of execution instances based on criticality and the merging, of schedules corresponding to these execution instances based on the combination of their criticality and probability offers better schedulability as compared to considering only criticality or probability. Currently, we are working on a model which adds fault-tolerance constraints to the object-based task model.

References

1. A. D. Stoyenko, L. R. Welch, J. P. C. Verhoosel, D. K. Hammer, and E. Y. Luit, "A model for scheduling of object-based, distributed real-time systems," *J. Real-Time Systems*, vol. 8, pp. 5-34, Aug. 1995.
2. B. W. Weide, W. F. Ogden, and S. H. Zweben. "Reusable software components," in M. C. Yovits, editor, *Advances in Computers*, Academic Press, vol. 33, pp. 1-65, 1991.
3. G. Yu, "Identifying and exploiting concurrency in object-based real-time systems," Ph.D. Thesis, New Jersey Institute of Technology, Jan. 1996.

4. H. El-Rewini and H. H. Ali, "Static scheduling of conditional branches in parallel programs," *J. Parallel and Distributed Computing*, vol. 24, pp. 41-54, Jan. 1995.
5. J. P. C. Verhoosel, D. K. Hammer, G. Yu, and L. R. Welch, "Pre-run-time scheduling for object-based, concurrent, real-time applications," *Proc. Workshop on Real-Time Applications*, Jul. 1994.
6. J. P. C. Verhoosel, L. R. Welch, D. K. Hammer, and E. J. Luit, "Incorporating temporal considerations during assignment and pre-run-time scheduling of objects and processes," *J. Parallel and Distributed Computing*, vol. 36, no. 1, pp. 13-31, Jul. 1996.
7. D. Towsley, "Allocating programs containing branches and loops within a multiple processor system," *IEEE Trans. Software Engg.*, vol. 12, no. 10, pp. 1018-1024, Oct. 1986.
8. T. C. K. Chou and J. A. Abraham, "Load balancing in distributed systems," *IEEE Trans. Software Engg.*, vol. 8, no. 4, pp. 401-412, Jul. 1982.
9. I. Santhoshkumar, G. Manimaran, and C. Siva Ram Murthy, "A pre-run-time scheduling algorithm for object-based distributed real-time systems," *Proc. IEEE Workshop on Parallel and Distributed Real-Time Systems,* Geneva, Apr. 1997.

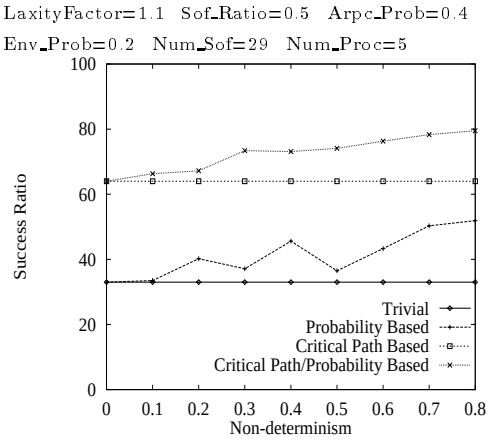

Figure 2 Effect of non-determinism

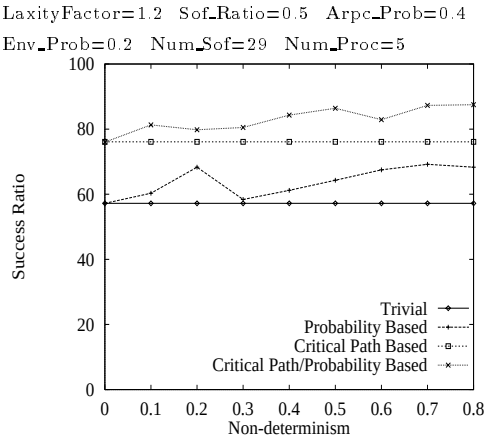

Figure 3 Effect of non-determinism

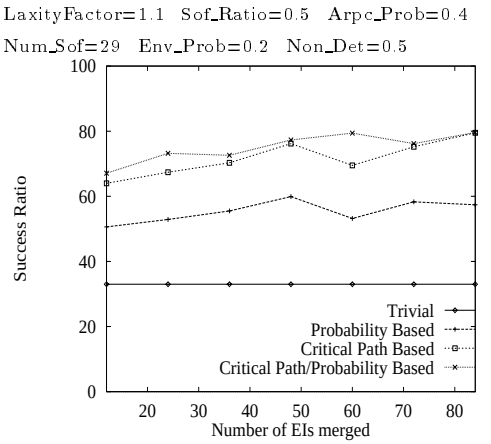

Figure 4 Effect of number of EIs merged

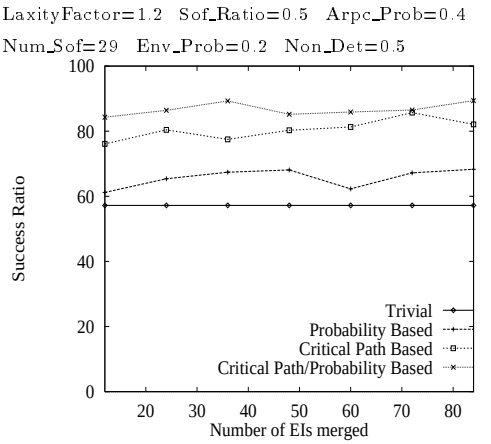

Figure 5 Effect of number of EIs merged